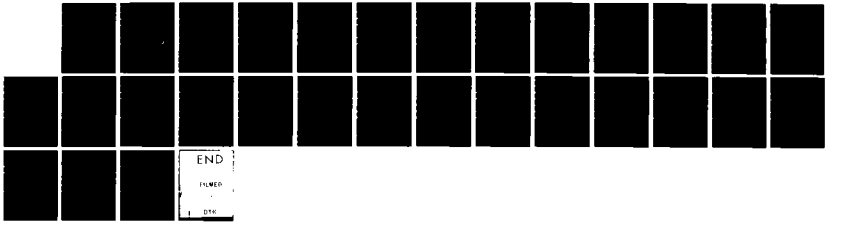
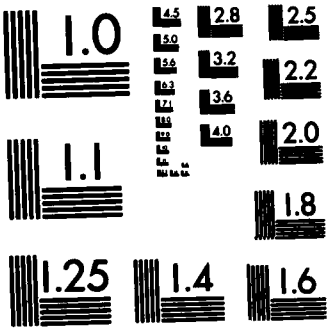
AD-A123 256 MRS MANUAL(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE 1/1
M R GENESERETH ET AL. OCT 81 HPP-80-24 N00014-81-K-0004

UNCLASSIFIED F/G 9/2 NL





Stanford Heuristic Programming Project
Memo HPP-80-24

First Version December 1980
Current Version October 1981

N00

MRS Manual

by

Michael R. Genesereth
Russell Greiner
David E. Smith

Department of Computer Science
School of Humanities and Sciences
Stanford University

Table of Contents

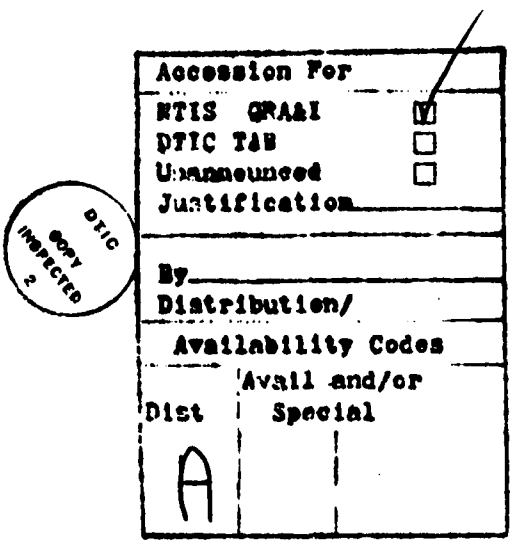
Chapter 1 - Introduction

MRS is a knowledge representation system intended for use by AI researchers in building expert systems. It offers a diverse repertory of commands for asserting and retrieving information, with various inference techniques (e.g. backward and forward chaining) and various search strategies (e.g. depth-first, breadth-first, and best-first search). The initial system includes a vocabulary of concepts and facts about logic, sets, mappings, arithmetic, and procedures. Additional "plug-in" modules are available to handle contexts, default reasoning, and truth maintenance. There is a rudimentary compiler to eliminate needless meta-level processing and a meta-level consistency checker to protect the user from fatal errors in making system modifications.

What makes MRS special among knowledge representation systems is its ability to utilize multiple representations while providing a single representation-independent language for stating facts. Because of its multiple representations, it is computationally superior to other knowledge representation systems; and, because of its powerful language and inference capability, it is expressively superior to traditional data definition languages

A second important feature is the system's meta-level capability. In MRS the system is treated as a domain in its own right. One can write sentences about subroutines and other sentences and allow the system to reason with them, just as it reasons about geology or medicine. Environmental considerations (like running time and storage requirements) and domain and range restrictions can easily be expressed. In practice, MRS uses this "meta-level" information in deciding how to carry out each operation. Thus, one can easily switch representations or inference methods by changing these sentences, and one can implement a variety of different meta-control schemes. Furthermore, the system is completely modifiable in that, by a progression of such changes, it can be converted into any program whatsoever.

Section 2 of this manual describes the syntax of MRS in detail, and section 3 lists the system's subroutines and variables. Section 4 presents the initial vocabulary; and section 5 lists the initial data base of facts about the predefined symbols. A summary of the system's utilities and the details of loading and using MRS are given in section 6.



Chapter 2 - The Syntax of MRS

MRS is a prefix version of the language of predicate calculus.

2.1 Symbols

There are two types of symbols in MRS, viz. *variables* and *constants.* Variables are useful for stating facts about all members of a set or for declaring the existence of an object without naming it. The use of variables is elaborated below in the discussion of quantified propositions.

There are three different types of constant symbols. *Object symbols* name specific objects or concepts in the world being described, e.g.

```
Stanford
Kennedy
elephants (the set thereof)
blue
justice
```

Function symbols are intended to represent functions on the objects of the world, e.g.

```
president-of
height-of
cardinality-of
sine
+
color-of
```

Relation symbols represent relations between objects of the world, e.g.

```
neighbor
subclass
>
older-than
```

2.2 Terms

In MRS one can also designate objects by combining these symbols into more complex expressions, called *terms.* All variables and constants are terms by definition. In addition, given an n-ary function symbol `f` and n terms `t1, . . ., tn`, then the expression `(f t1 . . . tn)` is also a term. For example, the following expressions are legal terms.

```
(president-of Stanford)
(cardinality-of elephants)
(+ 2 2)
(height-of (president-of Stanford))
(* (+ 2 2) 3)
```

2.3 Atomic Propositions

Facts can be stated withn MRS in the form of *propositions*. Given an n-ary relation symbol `r` and n terms `t1, . . . , tn`, the expression `(r t1 . . tn)` is an *atomic proposition*. The following are examples.

```
(neighbor Palo-Alto Menlo-Park)
(subclass elephants mammals)
(> (* 2 3) (+ 2 3)).
```

2.4 Logical Propositions

Unfortunately, not all facts are so simple. One often needs to express negations (e.g. "Lyman is not the president of Stanford), disjunctions (e.g. "Either Lyman is president or Kennedy is president"), and contingencies (e.g. "If George is at home, he must be sick"). In MRS facts like these can be written by relating the appropriate atomic propositions via *logical symbols* such as `not`, `and`, `or`, and `if`. For example, these sentences could be written as follows.

```
(not (= (president-of Stanford) Lyman))
(or (= (president-of Stanford) Lyman)
    (= (president-of Stanford) Kennedy))
(if (location george home) (sick george))
```

2.5 Quantified Propositions

Finally, there are quantified propositions. With the syntax given so far, one can only write facts by naming the objects involved. There's no simple way to talk about all the members of a set or state the existence of an object without naming it. Quantifiers enable one to state facts like "All apples are red" and "There's a doctor in the house". There are two quantifiers in MRS, viz. `all` and `exist`. The proposition `(all x1 . . . xn (p x1 . . . xn))` states that `(p x1 . . . xn)` is true for all possible values of the variable symbols `x1, . . . , xn`. The proposition `(exist x1 . . . xn (p x1 . . . xn))` states that there exist objects `a1, . . . , an` for which `(p a1 . . . an)` is true. For example, the first proposition below states that all apples are red, and the second says that there's a doctor in the house. Quantified propositions can also occur within non-atomic propositions, as in the last two examples.

```
(all x (if (mem x apples) (color-of x red)))
(exist x (and (mem x doctors) (location-of x house)))
(or (all x (apple x)) (some x (pear x)))
(all x (exist y (> y x)))
```

Multiple variables of the same type can be declared within a single quantified proposition, as illustrated below. Note, however, that the ordering of quantifiers is essential whenever a quantified proposition is nested within another. For example, the last two propositions mean two very different things. Information about the order of nesting of quantified propositions is sometimes referred to as *skolem information*.

```
(all h r (if (and (horse h) (rabbit r)) (can-outrun h r)))
(exist x y (and (= (+ x 1) y) (= (* 2 x) y)))
```

```
(all x (exist y (loves x y)))
(exist y (all x (loves x y)))
```

One particularly useful syntactic shorthand is the use of the prefix characters $ and ? to denote universal variables and leftmost existential variables. Each member of the following pairs of assertions is equivalent to the other.

```
(all x (if (neighbor x Bertram) (neighbor x Beatrice)))
(if (neighbor $x) (neighbor $x Beatrice))

(exist x (and (apple x) (color-of x red)))
(and (apple ?x) (color-of x red))
```

2.6 Higher Order Propositions

Higher order formulas are formed either by using functional or relational variables or by applying higher order relations to functional or relational constants. The induction axiom shown below is a simple example.

> Whenever 0 is in a set and n's membership in the set implies the membership of n + 1, then all integers are in the set.

```
(all s (if (and (s 0) (all n (if (s n) (s (+ n 1)))))
           (all n (s n))))
```

(Note that asserting such facts does not necessarily ensure that they will be used. Whether an axiom is used or not depends on the inference procedure the user has selected.)

Chapter 3 -MRS's Subroutines and Variables

MRS provides its user with several levels of subroutines for accessing and modifying its data base. One important property of the user-level subroutines is that the system reasons about each operation before carrying it out.

3.1 User Level Subroutines and Variables

Important note: In order to facilitate interaction with MRS, there is an additional sequence of commands that check the user's input for errors and reformat them as necessary. This preprocessing is separated from the basic routines described below so that they can be used in programs without sacrificing the time necessary to perform these error checks and reformatting. The names of these error-checking and reformatting commands are obtained by prefixing the names below with dollar signs ("$"). For example, (`$stash p`) removes all prefix quantifiers, records appropriate skolem information, and calls stash on the result. The commands `$unstash`, `$lookup`, `$assert`, `$unassert`, `$truep`, `$trueps`, etc. are analogously defined. Until one is thoroughly familiar with MRS, one should always use the "`$`" versions.

Currently, the "plural" functions return lists of all appropriate items. In subsequent versions of MRS, these explicit lists may be replaced with generator functions.

There are two series of user-level commands in MRS. Each of the commands in the first series (`stash`, `unstash`, `lookup`, . . .) does meta-level inference to determine how its argument is represented and then calls the resulting method without inference. Each of the commands in the second series (`assert`, `unassert`, `truep`, etc.) does meta-level inference to determine how much inference to perform in handling its argument and then calls the resulting inference method.

`stash` - (`stash <p>`)
: places proposition <p> in the data base, using the method indicated by its `MyToStash` property.

`unstash` - (`unstash <p>`)
: removes the assertion <p> from the data base, using the method indicated by its `MyToUnstash` property. Note this is *not* equivalent to asserting the negation of <p>.

`lookup` - (`lookup <p>`)
: checks whether the proposition <p> is asserted in the current data base, using the method indicated by its `MyToLookup` property. If <p> contains any existential variables, the result is a list of variable bindings that satisfies <p>. If <p> is free of existential variables and <p> is in the data base, the result is the default binding list ((t . t)).

`lookups` - (`lookups <p>`)
: checks whether the proposition <p> is present in the current data base, using

the method indicated by its MyToLookups property. If <p> contains any existential variables, the result is a list of binding lists that satisfy <p>. If <p> is free of existential variables and <p> is in the data base, the result is the list (((t . t))).

lookupval - (lookupval (<r> <x1> . . . <xn>))
uses the method indicated by its MyToLookupval property, which is usually equivalent to (subvar '$ (lookup (r x1 . . . xn ?))).

lookupvals - (lookupvals (<r> <x1> . . . <xn>))
uses the method indicated by its MyToLookupvals property, which is usually equivalent to (mapcar '(lambda (l) (subvar '?y l)) (lookups (r x1 . . . xn ?y))).

assert - (assert <p>)
places the proposition <p> in the data base and performs all appropriate forward inference, using the assertion method indicated by <p>'s MyToAssert property.

unassert - (unassert <p>)
runs the deletion method indicated by p's MyToUnassert property.

truep - (truep <p>)
tries to determine whether <p> is inferrable from the propositions in the data base, using the inference method indicated by p's MyToTruep property. If <p> contains any existential variables, the result is a list of variable bindings that satisfies <p>. If <p> is free of existential variables and <p> is in the data base, the result is the default binding list ((t . t))

why - (why <p>)
is identical to truep except that it saves justifications for each step in a deduction.

trueps - (trueps <p>)
runs the inference method indicated by p's MyToTrueps property. If <p> contains any existential variables, the result is a list of binding lists that satisfy <p>. If <p> is free of existential variables and <p> is in the data base, the result is the list (((t . t)))

getbdg - (getbdg <p>)
is equivalent to (cdar (truep p)), i.e. it assumes p contains exactly one existential variable and returns the value of that variable in the resulting binding list.

getbdgs - (getbdgs <p>)
is equivalent to (mapcar 'cdar (trueps p)), i.e. it assumes p contains exactly one existential variable and returns the values of that variable in each of the resulting binding lists.

`getval - (getval (<r> <x1> . . . <xn>))`
uses the method indicated by its MyToGetval property, which is usually equivalent to (getbdg (r x1 . . . xn ?y)).

`getvals - (getvals (<r> <x1> . . . <xn>))`
uses the method indicated by its MyToGetvals property, which is usually equivalent to (getbdgs (r x1 . . . xn ?y)).

3.2 Predefined Representational Methods

`pl-stash - (pl-stash (<f> <a> <b>))`
adds the proposition specified as its argument to the data base in property list format, i.e. `<b>` is placed on `<a>`'s property list under the indicator `<f>`. This representation is especially useful for representing the values of unary functions.

`pl-unstash - (pl-unstash (<f> <a> <b>))`
removes the proposition `<p>` from the property list representation by deleting the `<f>` property from the atom `<a>`.

`pl-lookup - (pl-lookup (<f> <a> <b>))`
checks whether the proposition `(<f> <a> <b>)` exists in the property list representation by matching `<b>` against the `<f>` property of the atom `<a>`.

`pl-lookupval - (pl-lookupval (<f> <a>))`
returns the `<f>` property of the atom `<a>`.

`dl-stash - (dl-stash (<r> <a> <b>))`
adds the proposition specified as its argument to the data base in "multiple value" format, i.e. it adds `<b>` to the list of values stored as the `<r>` property of the atom `<a>`. This representation is particularly useful for representing binary relations.

`dl-unstash - (dl-unstash (<r> <a> <b>))`
removes the proposition specified as its argument by removing `<b>` from the list of values stored as the `<r>` property of `<a>`.

`dl-lookup - (dl-lookup (<r> <a> <b>))`
checks whether the propostion specified as its argument is in the data base by matching `<b>` against each of the values stored as the `<r>` property of `<a>` until it either succeeds or runs out of possibilities.

`dl-lookups - (dl-lookups (<r> <a> <b>))`
checks whether the propostion specified as its argument is in the data base by matching `<b>` against all of the values stored as the `<r>` property of `<a>`.

`dl-lookupval - (dl-lookup (<r> <a>))`
returns the first element of the list stored as the `<r>` property of the atom `<a>`.

`dl-lookupvals - (dl-lookupvals (<r> <a>))`
returns the list of values stored as the `<r>` property of the atom `<a>`.

`pr-stash - (pr-stash <p>)`
adds the proposition `<p>` to the propositional data base and returns the corresponding proposition symbol.

`pr-unstash - (pr-unstash <p>)`
removes the proposition `<p>` from the propositional data base.

`pr-lookup - (pr-lookup <p>)`
checks whether the proposition `<p>` is in the propositional data base and, if so, returns the proposition symbol.

`pr-lookupval - (pr-lookupval (<f> <a1> . . . <an>))`
checks whether there is a proposition of the form (<f> <a1> . <an> <b>) in the propositional data base and, if so, returns `<b>`.

`pr-lookupvals - (pr-lookupvals (<f> <a1> . . . <an>))`
is similar to `pr-lookupval` except that it returns a list objects `<b>` satisfying `(<f> <a1> . . . <an> <b>)`.

`pr-getfacts - (pr-getfacts <n>)`
returns a list of all propositions containing the symbol `<n>` in the currently active contexts.

`ua-pattern - (ua-pattern <d>)`
returns the proposition corresponding to the proposition symbol `<d>`.

`ua-datum - (ua-datum <p>)`
returns the proposition symbol corresponding to the proposition `<p>`.

`ua-list - (ua-list t)`
returns a list of all symbols that occur in currently asserted propositions.

`ut-activate - (ut-activate <c1> . . . <cn>)`
makes the assertions in the theories `<c1> . . . <cn>` available for access.

`ut-deactivate - (ut-deactivate <c1> . . . <cn>)`
deactivates the named theories.

`ut-empty - (ut-empty <c>)`
unasserts all the facts in the theory `<c>`.

`currenttheory`
is the name of the currently writeable theory. Any atom is an acceptable theory name.

`activetheories`
is a list of all the currently active theories, those whose assertions are available for retrieval or deduction.

`ut-newtheory - (ut-newtheory <c1> . . . <cn>)`
generates a "garbage-collectible" theory with subtheories `<c1> . . . <cn>`. Currently, this works only in the Maclisp version.

`stash-indb - (stash-indb (indb <p>))`
stores the proposition specified as its argument in the data base by stashing the proposition `<p>`.

`unstash-indb - (stash-indb (indb <p>))`
unstashes the proposition specified as its argument by unstashing the proposition `<p>`.

`lookup-indb - (lookup-indb (indb <p>))`
checks whether the proposition specified as its argument is in the data base by calling `lookup` on the proposition `<p>`.

`stash-true - (stash-true (true <p> <s>))`
stores the proposition `<p>` in the context `<s>`.

`unstash-true - (unstash-true (true <p> <s>))`
removes the proposition `<p>` from the context `<s>`.

`lookup-true - (lookup-true (true <p> <s>))`
binds `currenttheory` to the context `<s>` and calls `truep` on `<p>`.

3.3 Predefined Inference Methods

`ex-truep - (ex-truep <p>)`
uses a matching procedure to determine whether p is implied by the data base. It matches universal variables in the data base and existential variables in p only. For example, the data base assertion `(r $x $x $y $x)` matches the pattern `(r a a (f $z) $z)` with the result `(($z . a) (t . t))`.

`bc-truep - (bc-truep <p>)`
uses backward chaining to determine whether p is true.

`cache`
is a variable governing whether bc-truep and ex-truep should "cache" their results. In using MRS's caching capability, it is recommended that one set `currenttheory` to some "scratch" theory like `cache` so that one can apply `ut-empty` when the cached values are no longer needed.

`bs-truep - (bs-truep <p>)`
is equivalent to `(or (lookup p) (bc-truep p))`. For example, if the data base contained the assertions `(if (mem $x birds) (flies $x))`, `(if (feathered $x) (mem $x birds))`, and `(feathered Herbie)`, then `(bs-truep (flies $z))` would return `(($z . Herbie) (t . t))`.

`pi-truep - (pi-truep <p>)`
uses property inheritance to determine whether p is true. For example, if the data base contained the assertions `(mem clyde elephants)`, `(subclass elephants mammals)`, and `(if (mem $x mammals) (temperature $x warm))`, then `(pi-truep (temperature clyde warm))` would return `((t . t))`. Currently, this works only in the Interlisp version.

`truep-not - (truep-not (not <p>))`
applies deMorgan's laws as necessary and calls `bs-truep` otherwise.

`thnot - (thnot (not <p>))`
calls `truep` to determine whether `<p>` is true and returns `nil` if it is and otherwise returns `((t . t))`.

`truep-and - (truep-and (and <p1> . . . <pn>))`
uses `truep` to find a binding list that makes each of the `<pi>` true.

`trueps-and - (trueps-and (and <p1> . . . <pn>))`
is the plural version of `truep-and`.

`truep-or - (truep-or (or <p1> . . . <pn>))`
uses `truep` to find if there is a binding list that makes at least one of the `<pi>` true.

`trueps-or - (trueps-or (or <p1> . . . <pn>))`
is the plural version of `truep-or`.

`truep-trueps - (truep-trueps <p>)`
uses `truep` to find a single binding list that makes `<p>` true an, if successful, returns a list of that binding list.

`truep-getval - (truep-getval (<f> <a1> . . . <an>))`
uses `truep` to find a proposition of the form `(<f> <a1> . . . <an> <b>)` and if successful returns `<b>`.

`trueps-getvals - (trueps-getvals (<f> <a1> . . . <an>))`
is the plural version of `truep-getval`.

`fc-assert - (fc-assert <p>)`
forward chains on all implications of the form `(if <p> <q>)`.

`fa-assert - (fa-assert <p>)`
forward chains on all implications of the form `(if (and . . <p> . .) <q>)`.

`fs-assert - (fs-assert <p>)`
first checks whether `<p>` is already in the data base. If not, `fs-assert` stashes `<p>` and then calls `fc-assert` and `fa-assert`.

`assert-and - (assert-and (and <p1> . . . <pn>))`
places the proposition specified as its argument in the data base by asserting each of the propositions `<pi>`.

`assert-iff - (assert-iff (iff <p> <q>))`
places the proposition specified as its argument in the data base by asserting the two propositions `(if <p> <q>)` and `(if <q> <p>)`.

Chapter 4 - Vocabulary

This section contains a vocabulary of useful terms for encoding information about logic, sets and mappings, arithmetic, and time. Most of the terms are mentioned in one or more of the assertions in the appendices or chapter 5 or are used by one or more of the inference procedures in chapter 3.

In creating new names, MRS users often observe the following conventions. Sets are usually named in the plural, e.g. resistors is the set of all resistors. Functions that apply to an arbitrary number of arguments are terminated with the symbol *, e.g. (union* s_1 s_2 s_3) is the union of the sets s_1, s_2, and s_3.

4.1 Logic

`=` - `(= <a> <b>)`
means that the symbols `<a>` and `<b>` are synonymous, i.e. they refer to the same object.

`not` - `(not <p>)`
means that the proposition `<p>` is false.

`and` - `(and <p1> . . . <pn>)`
means that the assertions `<p1> . . . <pn>` are all true.

`or` - `(or <p1> . . . <pn>)`
means that one or more of the assertions `<p1> . . . <pn>` is true.

`if` - `(if <p> <q>)`
means that whenever proposition `<p>` is true, proposition `<q>` is true.

`iff` - `(iff <p> <q>)`
is equivalent to `(and (if <p> <q>) (if <q> <p>))`.

`all` - `(all <x1> . . . <xn> <p>)`
means that `<p>` is true for all bindings of the variables `<x1> . . . <xn>`. Note that the "`$`" functions remove all quantifiers to facilitate matching. Skolem information is retained on the property lists of the variables in `<p>`.

`exist` - `(exist <x1> . . . <xn> <p>)`
means that there is some binding for the variables `<x1> . . . <xn>` that makes `<p>` true. Note that the "`$`" functions remove all quantifiers to facilitate matching. Skolem information is retained on the property lists of the variables in `<p>`.

4.2 Sets and Mappings

`empty`

is the empty set.

`set` - (set <a1> . . . <an>)
is the set consisting of the members <a1> . . . <an>.

`seq` - (seq <a1> . . . <an>)
is the sequence consisting of the members <a1> . . . <an>.

`mem` - (mem <a> <s>)
means that <a> is a member of the set <s>.

`subset` - (subset <a> <b>)
means that the set <a> is a subset of the set <b>.

`union*` - (union* <s1> . . . <sn>)
stands for <s1> ∪ . . . ∪ <sn>.

`inter*` - (inter* <s1> . . . <sn>)
stands for <s1> ∩ . . . ∩ <sn>.

`composition*` - (composition* <f1> . . . <fn>)
means the unary function composed of the unary functions <f1> . . . <fn>. For example, (composition* spouse mother).

`inverse` - (inverse <f>)
is the inverse function of <f>.

4.3 Arithmetic

`+` - (+ a_1 . . . a_n)

`-` - (- a b)

`*` - (* a_1 . . . a_n)

`>` - (> a b)

`<` - (< a b)

4.4 Meta-Level Vocabulary

`myto<g>` - (myto<g> <p> <f>)
means that the subroutine <f> is to be called in performing the action <g> on argument <p>. Each of MRS's user-level commands has associated with it a relation that specifies the subroutine to be used in carrying out that command. The relation is named by prefixing the command's name with `MyTo`, e.g. `MyToAssert`. There are similar relations for each of the commands in section 3.1.

`rel - (rel <p>)`
is the symbol in the relational position of the proposition `<p>`.

`arg - (arg <i> <p>)`
is the symbol in the `<i>`th argument position of the proposition `<p>`.

`val - (val <p>)`
is the symbol in the value position of the proposition `<p>`, i.e. the second argument.

`prop - (prop r` $a_1 \ . \ . \ . \ a_n$`)`
is the proposition symbol made up of the symbols specified.

`indb - (indb <p>)`
means that the proposition `<p>` is stored in the data base.

`true - (true <p> <s>)`
means that proposition `<p>` is true in situation `<s>`. This relation is especially useful for making statements about time.

`just - (just <q> <m> <p1> . . . <pn>)`
states that the justification for believing `<q>` is teh inference method `<m>` and the premises `<p1> . . . <pn>`.

`mytheory - (mytheory <p> <t>)`
states that the proposition `<p>` is stored in the teory `<t>`.

`subtheory - (subtheory <t1> <t2>)`
says that the theory `<t1>` is a subtheory of the theroy `<t2>`.

Chapter 5 - Initial Data Base

5.1 Definitions of the User-level Subroutines

The Maclisp definitions of the key user-level subroutines in MRS are as follows. The Interlisp versions are equivalent.

```
(defun stash (p) (kb 'MyToStash p))
(defun unstash (p) (kb 'MyToUnstash p))
(defun lookup (p) (kb 'MyToLookup p))
(defun lookups (p) (kb 'MyToLookups p))
(defun lookupval (x) (kb 'MyToLookupval x))
(defun lookupvals (x) (kb 'MyToLookupvals x))

(defun assert (p) (kb 'MyToAssert p))
(defun unassert (p) (kb 'MyToUnassert p))
(defun truep (p) (kb 'MyToTruep p))
(defun trueps (p) (kb 'MyToTrueps p))
(defun getval (x) (kb 'MyToGetval x))
(defun getvals (x) (kb 'MyToGetvals x))

(defun kb (g x)
  (let ((goals goals))
       (if (memsamep (setq g (list g x '$)) goals) nil
           (funcall (subvar '$ (or (ex-truep g) (bc-truep g)))
                    x))))
```

The version of `kb` in the system first checks the list goals to see whether the goal has occurred before and, if so, halts. The point of this check is to prevent infinite recursions when in the process of satisfying a goal, an identical subgoal is generated.

5.2 Initial Meta-assertions

MRS starts out with 100 or so meta-assertions. The curious user can type `(ut-contents 'global)` to see a list of all the assertions in the `global` theory or `($facts <s>)` to see all the facts that MRS knows about the symbol `<s>`.

Chapter 6 - Using MRS

Copies of MRS and its documentation can be obtained by writing to the Heuristic Programming Project at the following address.

Secretary
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, California 94305

Bugs and comments should be sent either to the above address or via Arpanet to CSD.Genesereth@Score or CSD.Smith@Score.

6.1 System Utilities

`$apropos` - `($apropos <string>)`
returns a list of all atoms containing `<string>`.

`$defunit` - `($defunit <name> <p1> . . . <pn>)`
allows one to bind together the set of propositions `<p1> . . . <pn>` associated with the symbol `<name>`. The effect of `$defunit` is the same as if the propositions were associated singly; its use is purely cosmetic.

`$facts` - `($facts <n>)`
prints out all assertions about `<n>` in the currently active theories. This works only in Maclisp.

`$just` - `($just <p>)`
prints out the justification for the proposition `<p>`. The user-level command `why` automatically creates justification links as it reasons.

`$load` - `($load <f>)`
loads a file `<f>` of propositions.

`ua-list` - `(ua-list t)`
returns a list of all symbols that occur in currently asserted facts.

`ua-edit` - `(edit <n>)`
invokes the LISP editor to permit the user to edit the assertions associated with `<n>`. This works in the Interlisp version only.

6.2 MRS in Maclisp

MRS is defined as a set of subroutines and data structures that can be loaded into any Maclisp program with the load command. On the Score machine, this can be done by typing the following line to any Maclisp program.

```
(load '|<CSD.MRS>MRS.load|)
```

6.3 MRS in Franz Lisp

As in Maclisp, MRS in Franz Lisp is defined as a set of subroutines and data structures that can be loaded with the load command. On the Diablo machine, this can be done by typing the following line to any Franz Lisp program.

```
(load '/usr/hpp-dart/dart/mrg/mrs.load)
```

6.4 MRS in Interlisp

On Score, MRS is also available in Interlisp as a sysout. It can be obtained by typing the following line to the monitor.

```
<CSD.MRS>MRS.exe
```

References

Genesereth, M. R.: "The Architecture of a Multiple Representation System", HPP-81-6, Stanford University Computer Science Department, May 1981.

Greiner, R.: "A Representation Language Language", HPP-80-9, Stanford University Computer Science Department, May 1980.

Smith, D. E.: "CORLL: A Storage and File Management System for Knowledge Bases, HPP-80-8, Stanford University Computer Science Department, April 1980.

Appendix I - A Brief Tour of MRS

This appendix presents a brief demonstration of some of the capabilities of MRS. The lines preceded by "=>" were typed by the user; the subsequent lines were typed by MRS.

```
=> ($assert '(color clyde grey))
   |(color clyde grey)|

=> ($truep '(color clyde grey))
   ((t . t))

=> ($getval '(color clyde))
   grey

=> ($truep '(color $x grey))
   (($x . clyde) (t . t))

=> ($assert '(all x (p x)))
   |(p $x)|

=> ($truep '(p $z))
   (($z . $x) (t . t))

=> ($unassert '(color clyde grey))
   |(color clyde grey)|

=> ($truep '(color clyde grey))
   nil

=> ($assert '(mem clyde elephants))
   |(mem clyde elephants)|

=> ($assert '(all x (if (mem x elephants) (color x grey))))
   |(if (mem $x elephants) (color $x grey))|

=> ($getval '(color clyde))
   grey

=> ($assert '(all x (if (and (mem x plants) (color x purple))
                        (poisonous x))))
   |(if (and (mem $x plants) (color $x purple)) (poisonous $x))|

=> ($assert '(all x (if (mem x mushrooms) (mem x plants))))
   |(if (mem $x mushrooms) (mem $x plants))|

=> ($assert '(mem phil mushrooms))
   |(mem phil mushrooms)|

=> ($assert '(color phil purple))
   |(color phil purple)|

=> ($truep '(poisonous phil))
   ((t . t))

=> (setq currenttheory 'earth)
```

```
   earth

=> ($assert '(color sky blue))
   |(color sky blue)|

=> (setq currenttheory 'mars)
   mars

=> ($assert '(color sky red))
   |(color sky red)|

=> ($getval '(color sky))
   red

=> (setq currenttheory 'earth)
   earth

=> ($getval '(color sky))
   blue

=> (setq currenttheory 'global)
   global

=> ($facts mytostash)
   (mytostash $x pr-stash)
   (mytostash (indb $p) stash-indb)
   (mytostash (subworld $c $d) stash-subworld)
   nil

=> ($assert '(mytostash (neighbor $x $y) dl-stash))
   |(mytostash (neighbor $x $y) dl-stash)|

=> ($assert '(mytolookupvals (neighbor $x) dl-lookupvals))
   |(mytolookupvals (neighbor $x) dl-lookupvals)|

=> ($assert '(neighbor stanford paloalto))
   (paloalto)

=> ($assert '(neighbor stanford menlopark))
   (menlopark paloalto)

=> (get 'stanford 'neighbor)
   (menlopark paloalto)

=> ($lookupvals '(neighbor stanford))
   (menlopark paloalto)

=> ($facts not)
   (mytotruep (not $p) truep-not)
   (mytotrueps (not $p) trueps-not)
   nil

=> ($truep '(not (striped zeke)))
   nil

=> ($assert '(mytotruep (not $x) thnot))
   |(mytotruep (not $x) thnot)|

=> ($truep '(not (striped zeke)))
```

```
   ((t . t))

=> ($assert '(striped zeke))
   |(striped zeke)|

=> ($truep '(not (striped zeke)))
   nil

=> ($assert '(mother bertram allison))
   |(mother bertram allison)|

=> ($assert '(spouse allison arthur))
   |(spouse allison arthur)|

=> ($getval '(father bertram))
   nil

=> ($assert '(mytotruep (father $x $y) truep-father))
   |(mytotruep (father $x $y) truep-father)|

=> (defun truep-father (p)
     ($truep (list 'and
                   (list 'mother (cadr p) '$z)
                   (list 'spouse '$z (caddr p)))))
   truep-father

=> ($truep '(father bertram arthur))
   (($z . allison) (t . t))

=> ($getval '(father bertram))
   arthur

=> ($why '(poisonous phil))
   ((t . t))

=> ($just '(poisonous phil))
   (poisonous phil) by bc-truep
          (if (and (mem $x plants) (color $x purple)) (poisonous $x))
          (and (mem phil plants) (color phil purple))

   (and (mem phil plants) (color phil purple)) by truep-and
          (mem phil plants)
          (color phil purple)

   (mem phil plants) by bc-truep
          (if (mem $x mushrooms) (mem $x plants))
          (mem phil mushrooms)
   nil

=> ($facts phil)
   (mem phil mushrooms)
   (color phil purple)
   nil

=> ($facts mytoassert)
   (mytoassert $x stash)
   (mytoassert (and $p $q) assert-and)
   (mytoassert (iff $p $q) assert-iff)
   (mytoassert (true $p $s) assert-true)
```

```
   nil

=> ($assert '(mytoassert $p fs-assert))
   |(mytoassert $p fs-assert)|

=> ($assert '(mem joe mushrooms))
   |(mem joe mushrooms)|

=> ($assert '(color joe purple))
   |(color joe purple)|

=> ($facts joe)
   (mem joe mushrooms)
   (mem joe plants)
   (color joe purple)
   (poisonous joe)
   nil
```

Appendix 2 - An Example of Using MRS to Describe Digital Circuits

This appendix presents an example of how MRS can be used to describe the structure and behavior of digital circuits. The full adder was taken from Mano's book *Computer System Architecture.* The gate descriptions are oversimplifications in that temporal information has been omitted and a simple two-valued logic has been assumed. For a more extended discussion of how MRS can be used in describing circuits, the reader should see the "DDL Manual" [Genesereth, Grinberg, Lark]. Section A2.1 presents the description, and section A2.2 shows how it can be used in answering questions and generating explanations.

A2.1 The Description

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;             the behavioral description of an inverter            ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $g inv) (on (input 1 $g)))
    (off (output 1 $g)))

(if (and (type $g inv) (off (input 1 $g)))
    (on (output 1 $g)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;             the behavioral description of an and-gate            ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $g and-gate) (on (input 1 $g)) (on (input 2 $g)))
    (on (output 1 $g)))

(if (and (type $g and-gate) (off (input 1 $g)))
    (off (output 1 $g)))

(if (and (type $g and-gate) (off (input 2 $g)))
    (off (output 1 $g)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;             the behavioral description of an or-gate             ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $g or-gate) (or (on (input 1 $g)) (on (input 2 $g))))
    (on (output 1 $g)))

(if (and (type $g or-gate) (off (input 1 $g)) (off (input 2 $g)))
    (off (output 1 $g)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;             the behavioral description of an xor-gate            ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(if (and (type $g xor-gate) (off (input 1 $g)) (off (input 2 $g)))
    (off (output 1 $g)))

(if (and (type $g xor-gate) (off (input 1 $g)) (on (input 2 $g)))
```

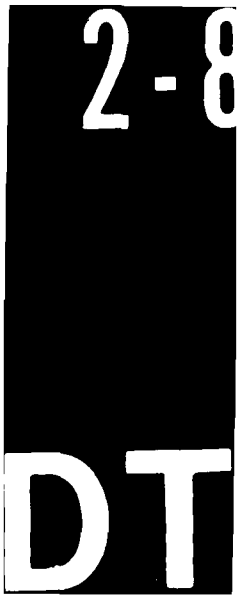
```
    (on (output 1 $g)))

(if (and (type $g xor-gate) (on (input 1 $g)) (off (input 2 $g)))
    (on (output 1 $g)))

(if (and (type $g xor-gate) (on (input 1 $g)) (on (input 2 $g)))
    (off (output 1 $g)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    Structural Description of a full adder. See Mano page 20.     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(prototype fa-1 full-adder)
(sizein fa-1 3)
(sizeout fa-1 2)

(subpart* xor-1 and-1 xor-2 and-2 or-1 fa-1)
(type xor-1 xor-gate)
(type and-1 and-gate)
(type xor-2 xor-gate)
(type and-2 and-gate)
(type or-1 or-gate)

(conn* (input 1 fa-1) (input 1 xor-1) (input 1 and-1))
(conn* (input 2 fa-1) (input 2 xor-1) (input 2 and-1))
(conn* (input 3 fa-1) (input 1 xor-2) (input 1 and-2))

(conn* (output 1 xor-1) (input 2 xor-2) (input 2 and-2))
(conn* (output 1 and-1) (input 1 or-1))
(conn* (output 1 and-2) (input 2 or-1))

(conn* (output 1 xor-2) (output 1 fa-1))
(conn* (output 1 or-1) (output 2 fa-1))
```

A2.2 Using the Description

```
=> ($assert '(on (input 1 fa-1)))
   |(on (input 1 fa-1))|

=> ($assert '(on (input 2 fa-1)))
   |(on (input 2 fa-1))|

=> ($assert '(off (input 3 fa-1)))
   |(off (input 3 fa-1))|

=> ($truep '(on (output $n fa-1)))
   (($n . /2) (t . t))

=> ($truep '(off (output $n fa-1)))
   (($n . /1) (t . t))

=> ($why '(on (output 2 fa-1)))
   ((t . t))

=> ($just '(on (output 2 fa-1)))
   (on (output 2 fa-1)) by bc-truep
       (if (and (conn $x $y) (on $x)) (on $y))
       (and (conn (output 1 or-1) (output 2 fa-1)) (on (output 1 or-1)))
```

```
(and (conn (output 1 or-1) (output 2 fa-1)) (on (output 1 or-1)))
    by truep-and
    (conn (output 1 or-1) (output 2 fa-1))
    (on (output 1 or-1))

(on (output 1 or-1)) by bc-truep
    (if (and (type $g or-gate)
             (or (on (input 1 $g)) (on (input 2 $g))))
        (on (output 1 $g)))
    (and (type or-1 or-gate)
         (or (on (input 1 or-1)) (on (input 2 or-1)))))

(and (type or-1 or-gate)
     (or (on (input 1 or-1)) (on (input 2 or-1)))) by truep-and
    (type or-1 or-gate)
    (or (on (input 1 or-1)) (on (input 2 or-1)))

(or (on (input 1 or-1)) (on (input 2 or-1))) by truep-or
    (on (input 1 or-1))

(on (input 1 or-1)) by bc-truep
    (if (and (conn $x $y) (on $x)) (on $y))
    (and (conn (output 1 and-1) (input 1 or-1))
         (on (output 1 and-1)))

(and (conn (output 1 and-1) (input 1 or-1)) (on (output 1 and-1)))
    by truep-and
    (conn (output 1 and-1) (input 1 or-1))
    (on (output 1 and-1))

(on (output 1 and-1)) by bc-truep
    (if (and (type $g and-gate) (on (input 1 $g)) (on (input 2 $g)))
        (on (output 1 $g)))
    (and (type and-1 and-gate)
         (on (input 1 and-1))
         (on (input 2 and-1)))

(and (type and-1 and-gate) (on (input 1 and-1)) (on (input 2 and-1)))
    by truep-and
    (type and-1 and-gate)
    (on (input 1 and-1))
    (on (input 2 and-1))

(on (input 1 and-1)) by bc-truep
    (if (and (conn $x $y) (on $x)) (on $y))
    (and (conn (input 1 fa-1) (input 1 and-1)) (on (input 1 fa-1)))

(and (conn (input 1 fa-1) (input 1 and-1)) (on (input 1 fa-1)))
    by truep-and
    (conn (input 1 fa-1) (input 1 and-1))
    (on (input 1 fa-1))

(on (input 2 and-1)) by bc-truep
    (if (and (conn $x $y) (on $x)) (on $y))
    (and (conn (input 2 fa-1) (input 2 and-1)) (on (input 2 fa-1)))

(and (conn (input 2 fa-1) (input 2 and-1)) (on (input 2 fa-1)))
    by truep-and
```

```
    (conn (input 2 fa-1) (input 2 and-1))
    (on (input 2 fa-1))

nil
```


2-8
DT